Introduction to the Operating System

HMM-155-A
CRAY T3E Series Systems
Last Modified: November 1997

Record of Revision

December 1996

Original printing.

Revision A: November 1997

This revision updates the information about the UNICOS/mk servers.

Introduction to the UNICOS/mk Operating System

The CRAY T3E operating system (OS) is a microkernel-based OS called UNICOS/mk OS. The CHORUS system provides the model for restructuring the UNICOS operating system into a microkernel-based system. This distributed OS design supports integrated systems that are composed of MPP systems and PVP systems.

For the CRAY T3E system, UNICOS/mk is physically distributed across the PEs. Each user PE contains a microkernel and a varying number of servers that distribute the OS.

The design goals for UNICOS/mk include:

- Support for the new Cray Research hardware architectures:
  - PVP
  - MPP
  - PVP clusters
  - Hybrid systems
- UNICOS compatibility
- Single system image (SSI)
- Increased performance
- Modularity

Terms

Software designers used the CHORUS system as the model for restructuring the UNICOS OS into the microkernel-based system. You will need to know the following CHORUS terms to fully understand the material discussed in this document. The actor, thread, port, and memory region terms are referred to as CHORUS abstractions.

Actor - An actor is composed of threads, ports, and memory regions. The UNICOS/mk servers and user processes are actors.

Application - An application is a multi-PE program.

Capability - A capability is a general class of structures generated to facilitate quick access during subsequent references to an object (e.g. file structures).

Credentials - Credentials are security-type information (for example, user ID, permission bits, etc.) that indicate whether the requester has permission to access a file.

Memory Region - A memory region is a portion of memory in the actor's address space.

Port - A port is an address where actors send messages. A port is also a queue of unread messages. When a port is created, it is attached to a specific actor and only threads of this actor can receive messages on that port.

Registration - Registration occurs during the initialization of a command. During this time, the software indicates the addresses of all functions that are associated with the servers.

Thread - A thread represents a unit of sequential execution.

User process - A user process is an instance of a program. When a user process runs in one PE, it is referred to as a command.

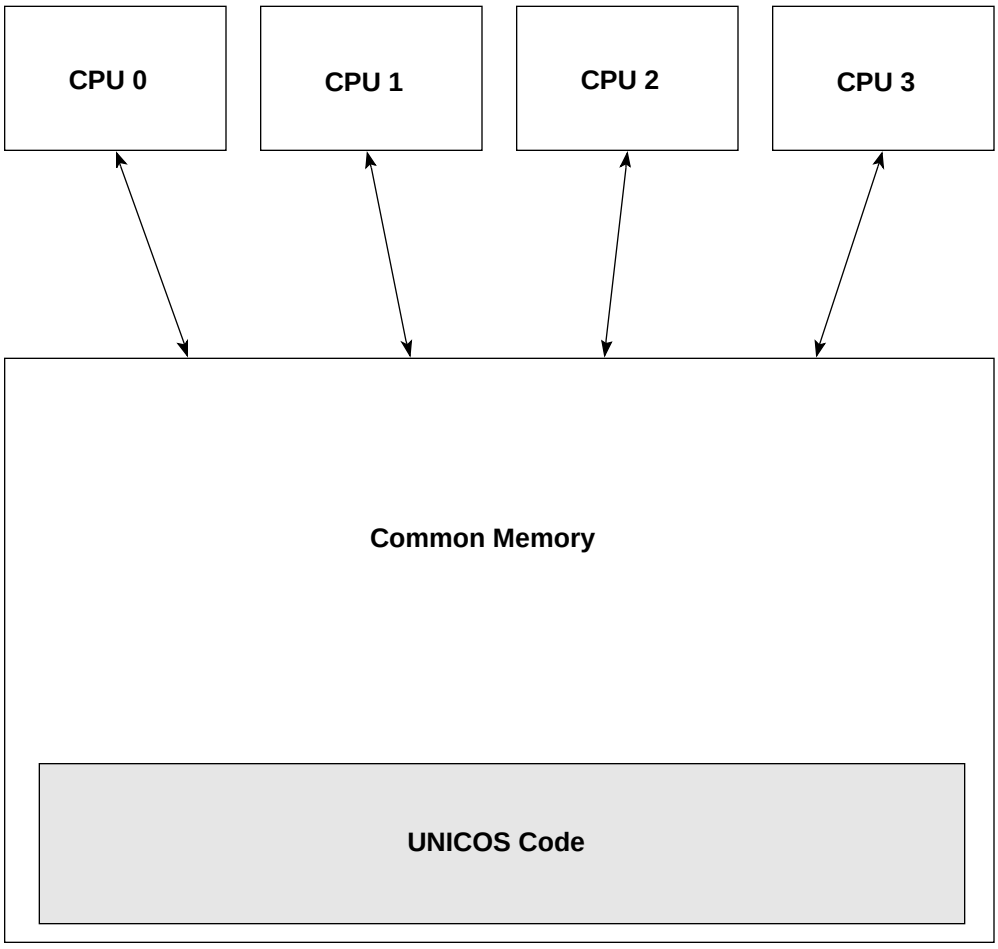
Wrapper - A wrapper is additional code applied to existing UNICOS code that allows it to function as a server in the UNICOS/mk environment.

Comparison of UNICOS to UNICOS/mk

UNICOS

UNICOS is compatible with systems that use a common-memory architecture. Within these systems, the entire UNICOS code resides in common memory. When a system contains multiple central processing units (CPUs), each CPU has equal access to common memory. In addition, each CPU requires access to the complete set of UNICOS code. Figure 1 shows an overview of UNICOS code in a common-memory architecture.

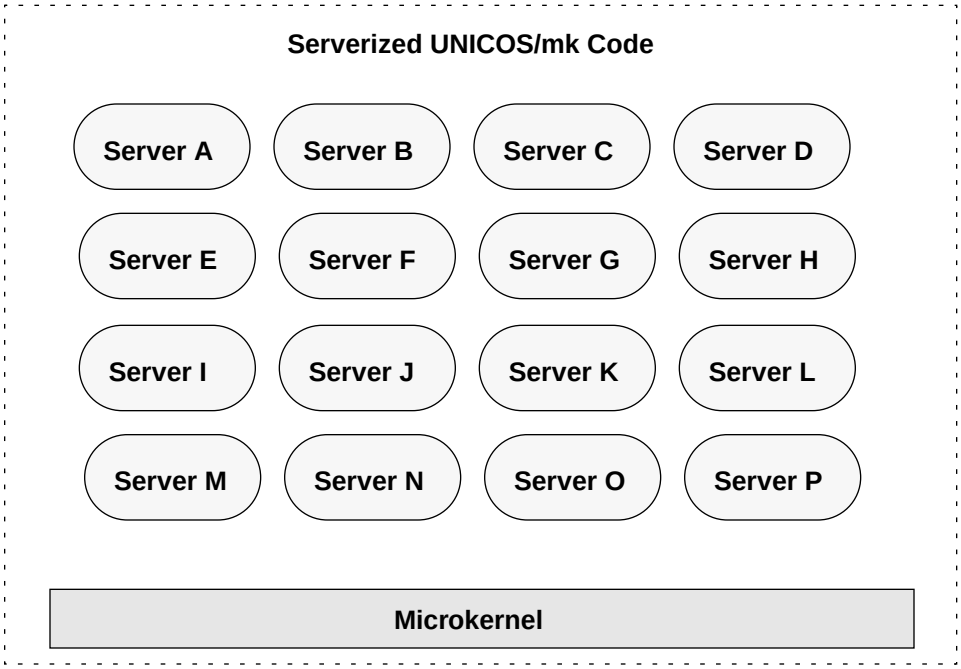
Figure 1. UNICOS Code in a Common-memory Architecture



UNICOS/mk

UNICOS/mk does not require a common-memory architecture. Unlike UNICOS, the functions of UNICOS/mk are divided between a microkernel and numerous servers (refer to Figure 2). For this reason, UNICOS/mk is referred to as a serverized operating system.

Figure 2. Overview of Serverized UNICOS/mk



Serverized operating systems offer a distinct advantage for the CRAY T3E system because of its distributed memory architecture. Within these systems, the local memory of each PE is not required to hold the entire set of OS code. Instead, the operating system can be distributed across the PEs in the system.

Support PEs, User PEs, and Redundant PEs

In the CRAY T3E systems, the local memory of each PE must contain a copy of the microkernel and one or more servers. Under UNICOS/mk, each PE in a CRAY T3E system is configured as one of the following types of PEs:

- Support PEs

  The local memory of support PEs contains a copy of the microkernel and several servers. The exact number and type of servers vary depending on configuration tuning.

- User PEs

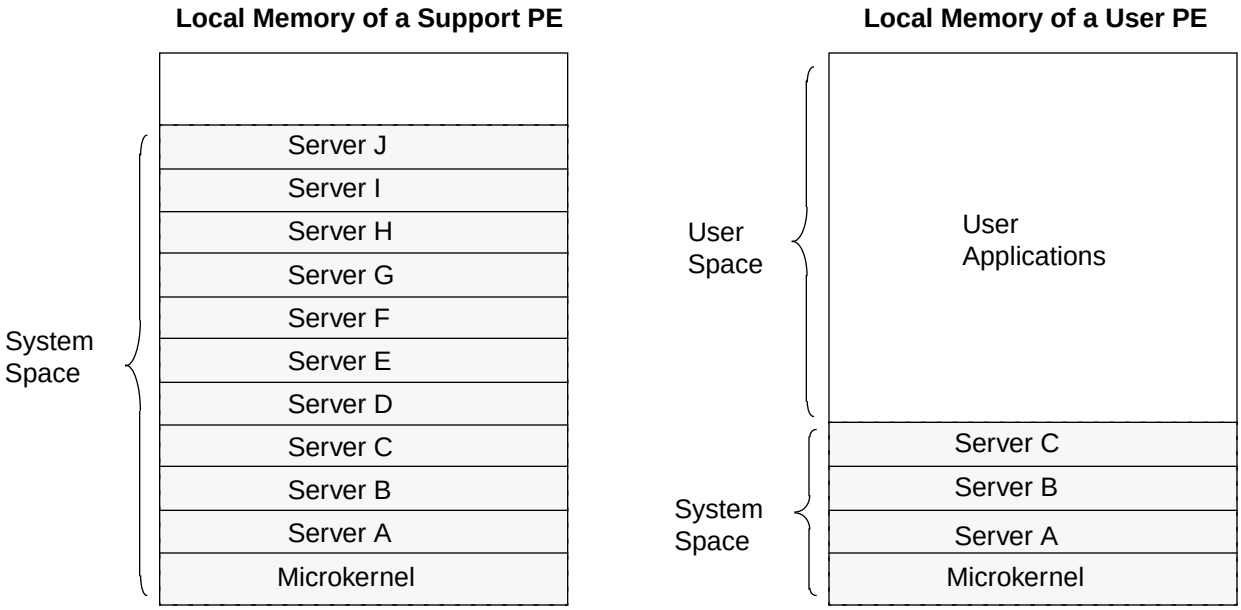
  The local memory of user PEs contains a copy of the microkernel and a minimum number of servers. Because it contains a limited amount of operating system code, most of a user PE's local memory is available to the user (refer to Figure 3). User PEs include command PEs and application PEs.

  - Command PEs handle single-PE user applications and commands.
  - Application PEs handle multiple-PE user applications.

- Redundant PE

  A redundant PE is not configured into the system until an active PE fails.

Figure 3. Example of UNICOS/mk Distribution in the Support PEs and User PEs

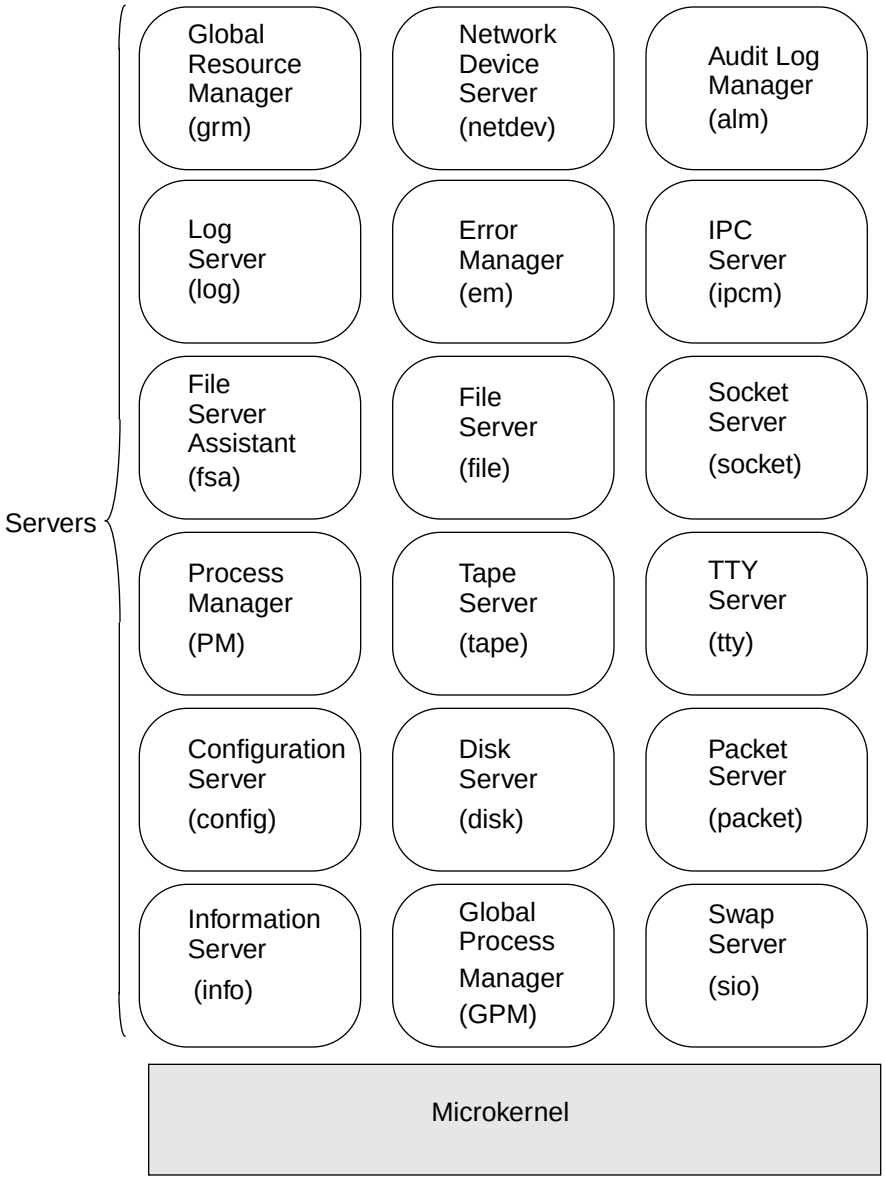


Overview of UNICOS/mk Architecture

UNICOS/mk is modularized, which means that the OS functions through a microkernel and servers that are distributed across multiple interconnected CRAY T3E PEs. Each PE contains a copy of the microkernel and its underlying mkPAL.

Figure 4 illustrates the major components of the UNICOS/mk architecture. The actual number of servers is not limited to those shown.

Figure 4. UNICOS/mk Architecture Overview



Microkernel (mk)

The microkernel supports a minimal set of primitives and the specifics of the logic design. A copy of the microkernel resides in the lower memory area of each PE in a CRAY T3E system. In a distributed OS, each occurrence of the microkernel is also referred to as a site.

The microkernel handles:

- Specifics of the logic design
- Memory management
- Interprocess communication (IPC)
- CPU scheduling (based on priorities set by a server-based scheduler)

Microkernel Privileged Architecture Library (mkPAL)

The microkernel handles the specifics of the logic design through a self-contained microkernel privileged architecture library (mkPAL).

The mkPAL provides two primary functions:

1. Insulate system and user code from the specifics of the logic design of the PE. Specifics of the logic design include:

   - Interrupt handling
   - User/kernel context switching
   - Virtual address translation

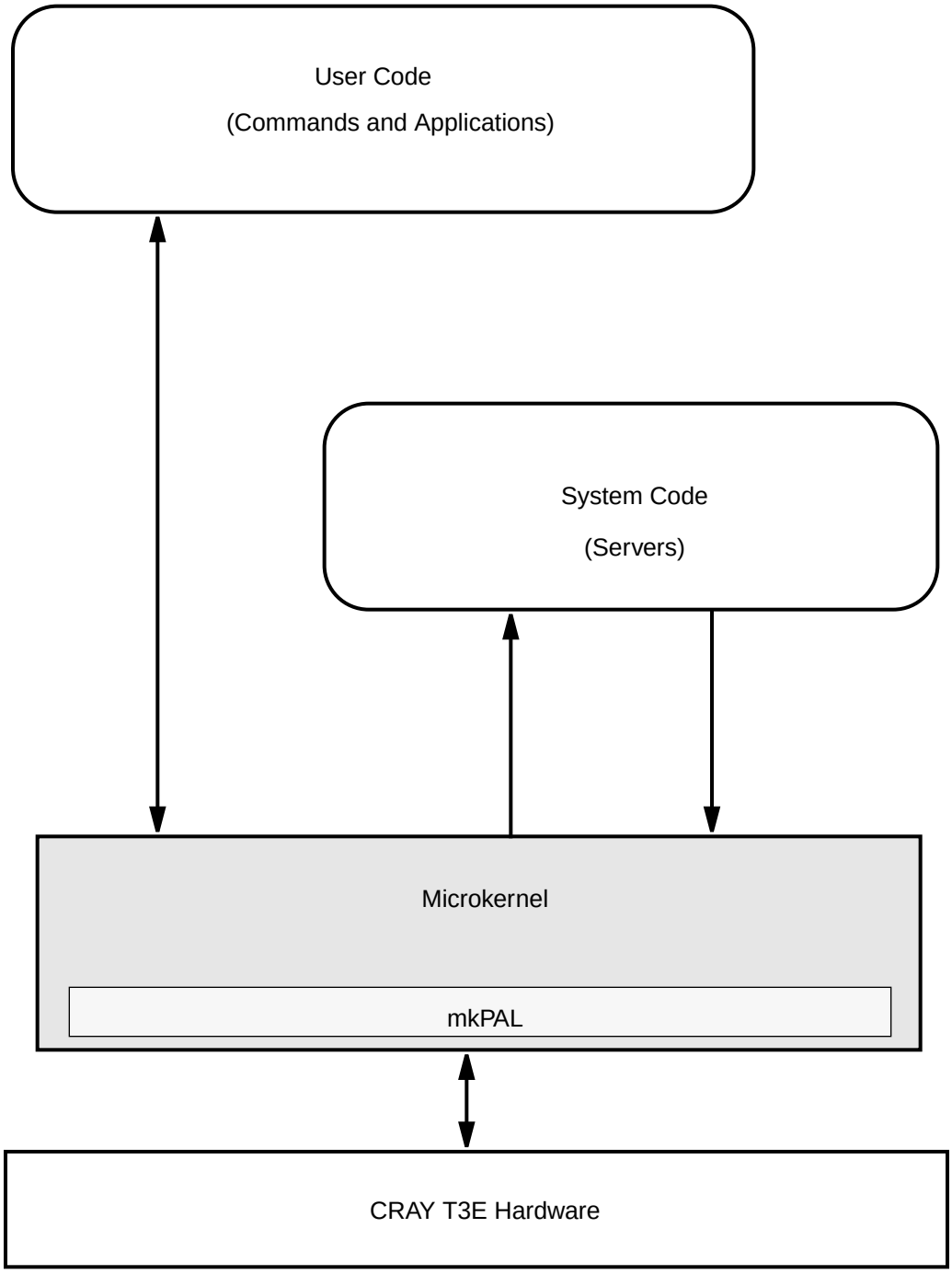
2. Interface system and user code with the hardware characteristics of the PE. Example interfaces include:

   - Exception handling
   - Special instructions such as system calls

A system call originates as an interrupt that causes a context switch to the microkernel. The microkernel passes the system call to the appropriate server(s). Subsequent communication from the server(s) to the microkernel occurs through microkernel interface functions.

Figure 5 highlights the microkernel in the CRAY T3E system.

Figure 5. Microkernel within UNICOS/mk



Microkernel Abstractions

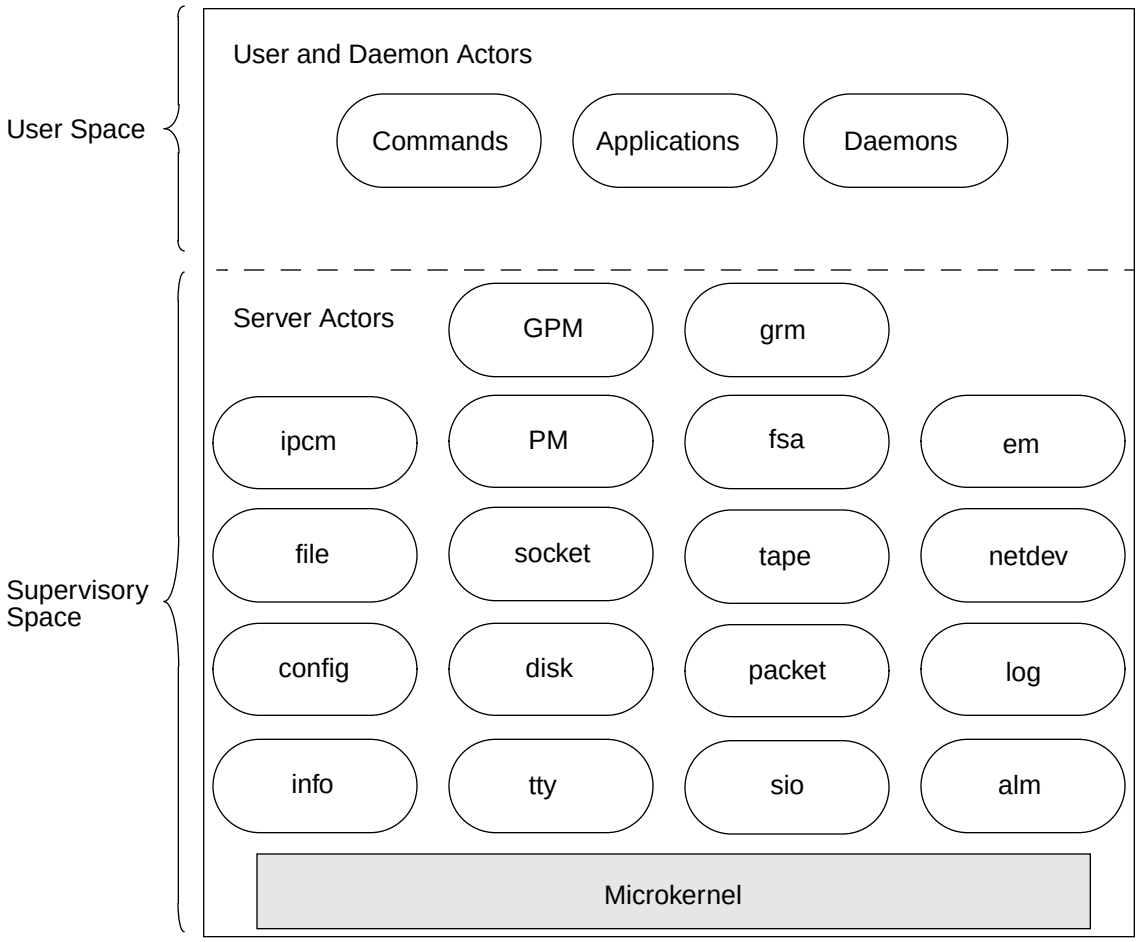
The microkernel supports basic abstractions that the servers use to provide OS functions. These basic abstractions include actors, threads, ports, and regions.

Actors

Under UNICOS/mk, traditional UNICOS processes are implemented as actors. Actors represent a resource allocation entity. The microkernel views all user processes, servers, and daemons as actors.

A multiple-PE application has one actor per PE. User and daemon actors reside in user address space; server actors reside in supervisory (kernel address) space. Figure 6 illustrates the location of actors in memory space.

Figure 6. Actors in Global Memory Space



Threads

A thread represents a resource scheduling entity and maintains a sequential flow of execution with a given microprocessor state. The microkernel uses threads to maintain the context of processes as they are switched in or out of a microprocessor. In addition, a thread contains structures that maintain its own context.

For example, when the microkernel switches a process out of a microprocessor, the microkernel retains and stores values from the program counter, stack pointer, and certain registers. When this process is switched back into the microprocessor, the microkernel reloads the stored values to maintain a sequential flow of execution.

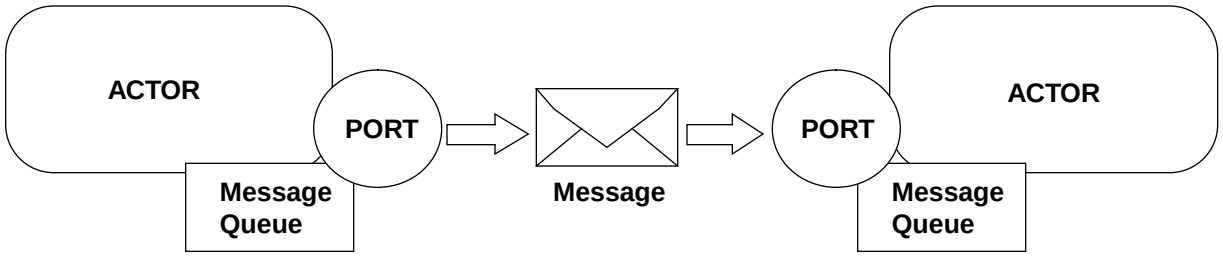
For a single-thread process, the microkernel creates the thread when the operating system allocates resources for the process; the microkernel deletes the thread when the process completes. For a multithread process, the microkernel creates a thread when the operating system allocates resources for the process; the microkernel can create or delete additional threads dynamically during the life of the process. A thread is always tied to a specific actor and executes within that actor's address space. Scheduling of functions within the microkernel is also based on threads.

Ports

A port is a unique reference address that facilitates location-independent communication between actors. Each port is attached to a specific actor and is supported by an associated message queue (refer to Figure 7). Ports also contain a function that processes incoming messages (message handler).

Message passing between ports is referred to as interprocess communication (IPC). A port may be a member of a port group, which is useful for broadcasting messages to multiple actors.

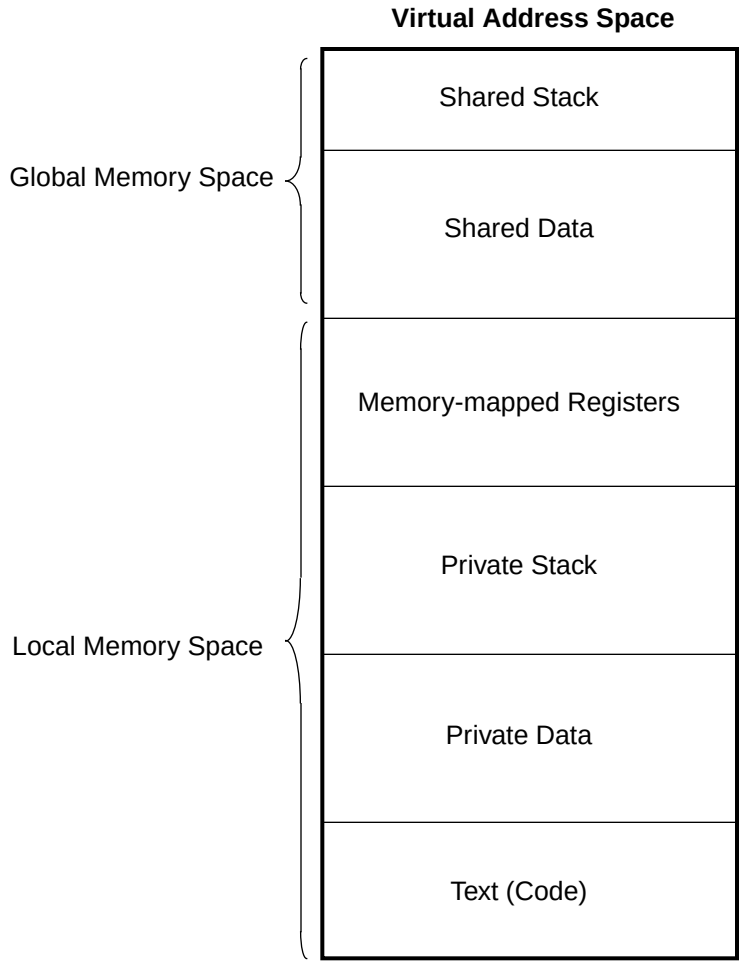
Figure 7. Port Example



Regions

A region is a microkernel structure representing a memory segment that describes the memory space of a kernel or an actor. Types of memory segments include text, data, stack, and memory-mapped register (MMR) segments (refer to Figure 8).

Figure 8. Memory Regions



UNICOS/mk Servers

A UNICOS/mk server is a statically linked executable file (for example, a.out). Each UNICOS/mk server:

- Contains a port to facilitate communication between servers
- Contains a service handler from a server library (SERVLIB)
- Sends and receives service calls from a service list (servCode.h)

Many of the UNICOS/mk servers are UNICOS system codes that are *wrapped* (code is added to UNICOS code so that it can function in a UNICOS/mk environment) in server-specific code and shared code. This use of existing code reduced development costs and contributed to UNICOS compatibility. Servers implement wrapper code to provide:

- A messaging interface that is specific to each server
- UNICOS kernel emulation

Figure 9 illustrates a wrapper with a UNICOS server. Servers that are developed specifically for UNICOS/mk (were not ported from UNICOS code) do not require wrappers.

Figure 9. Example of a Server Wrapper

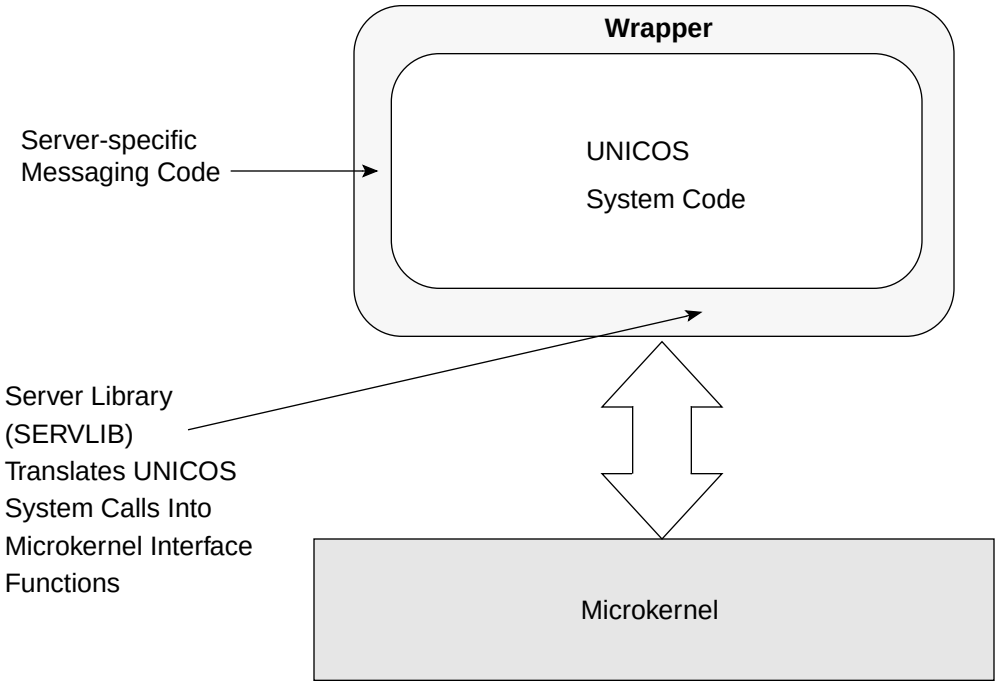


Table 1 lists and describes some of the UNICOS/mk servers.

Table 1. UNICOS/mk Servers

Server	Identifier	Description	Usage Rules
Global process manager	GPM	Manages process connections that involve more than one PE	Boot PE
Process manager	PM	Provides the operating system interface to an application	All PEs
Global resource manager	grm	Tracks resource availability and allocates resources to user jobs and applications; performs the equivalent of UNICOS sched functions	Boot PE
Error manager	em	Detects and provides resiliency when reported hardware and software errors occur	All PEs
Configuration server	config	Processes and updates configuration information	Boot PE
Disk server	disk	Receives disk-related I/O requests and accesses the appropriate device drivers	One or more OS PEs; this server connects to a packet server
File server	file	Provides a file system interface to the application and makes requests to disk servers for data	One or more OS PEs; one file server must be on the same PE as the socket server for the NFS
File server assistant	fsa	Implements part of the functionality of the file server; allows local handling of some read and write operations to reduce IPC latency	All user PEs
Information server	info	Receives requests for system (kernel) information and passes them to the appropriate servers	Boot PE
Log server	log	Receives requests to log messages from other system servers	Boot PE
Swap server	sio	Allocates space on the swap device	One OS PE
Network device server	netdev	Forwards network device requests from the socket server to the packet server	One OS PE that connects to a packet server and is on the same PE as the socket server
Packet server	packet	Provides both a messaging and direct call interface to other servers that send and receive packets on the GigaRing channel	One or more OS PEs; each packet server connects the disk, tty, netdev, and tape servers to one or more GigaRing channels. **NOTE:** There must only be one packet server per I/O controller.

Table 1. UNICOS/mk Servers (continued)

Server	Identifier	Description	Usage Rules
Socket server	socket	Accepts any socket operation from the process manager and forwards it to the network device server	One OS PE that is on the same PE as the network device server
Tape server	tape	Supports the UNICOS tape devices	One OS PE that connects to a packet server
tty server	tty	Provides the interface to the process manager and file server for tty character device operations	Boot PE that connects to a packet server
Audit log manager	alm	Logs accounting information	Boot PE
IPC server	ipcm	Facilitates IPC messages	Boot PE
Single-user mode init()	UP_INIT	Initializes the kernel to single-user mode	Boot PE

Distribution of Servers

The distribution of UNICOS/mk servers in the CRAY T3E systems is determined by several factors including:

- The functions that each server provides
- The frequency of communication between servers

Server Distribution in Support PEs

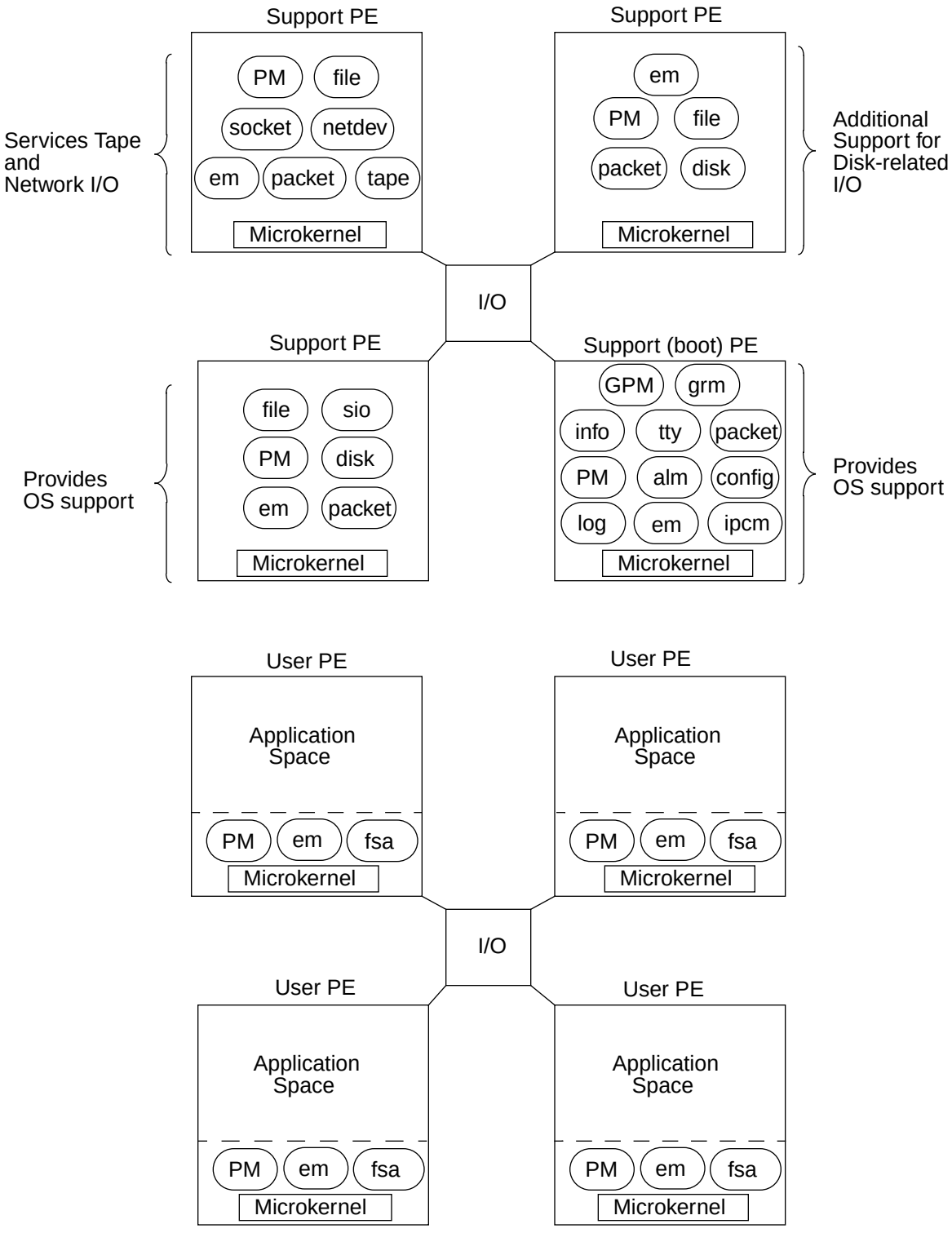
The UNICOS/mk architecture supports many options for the number of servers and the types of servers that a support PE can contain. The maximum number of servers that a support PE can contain is limited only by the storage capacity of the PE's local memory.

The types of servers that a support PE contains may be selected for performance reasons. For example, a user disk request requires a file server, a disk server, and a packet server. Grouping these servers on one support PE increases system performance because they can communicate within one site (refer to Figure 10).

Server Distribution in User PEs

Typically, a user PE contains a process manager (PM), a file server assistant, and an error manager. The PM server provides user applications with an interface to the operating system. The file server assistant performs some of the file server functions locally. The error manager detects and reports errors. Because user PEs contain a minimum number of servers, a large amount of memory space is available for user applications (refer again to Figure 10).

Figure 10. Example of Server Distribution in a CRAY T3E System



Interprocess Communication (IPC)

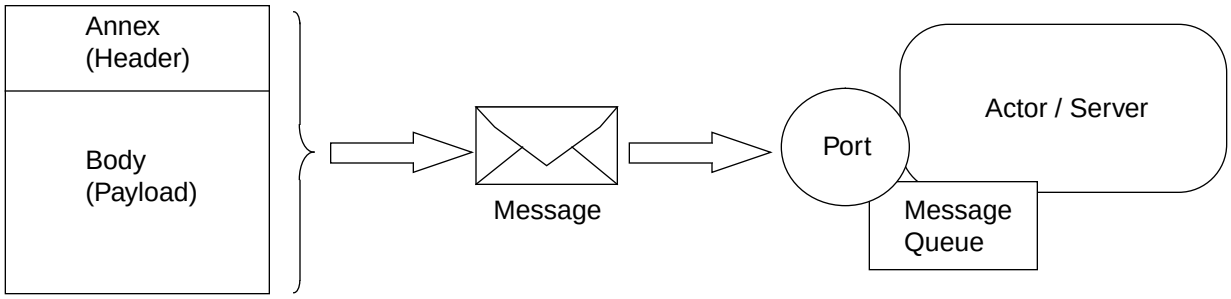
Interprocess communication (IPC) is the method that UNICOS/mk uses to send and receive messages. A message can be directed to a specific port address (one server) or broadcasted to a port group (several related servers).

The content of a message is defined by software and consists of:

- An annex (header)
- A body (payload) of variable but limited size

Figure 11 illustrates the components of an IPC message.

Figure 11. Components of an IPC Message



IPC operates in one of the following two modes:

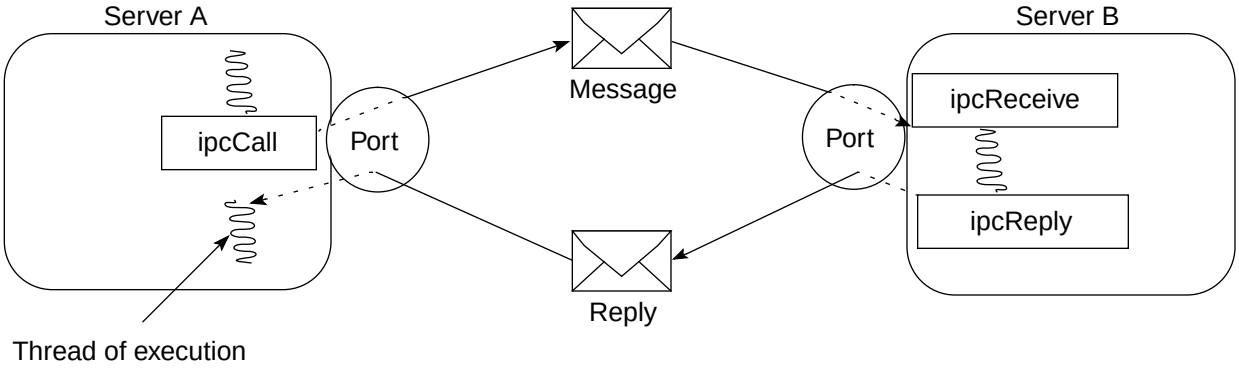
- Synchronous mode (implemented through ipcCall)

  Synchronous IPC requires that the receiving server reply to the sender (caller); the sender waits for the reply (refer to Figure 12).

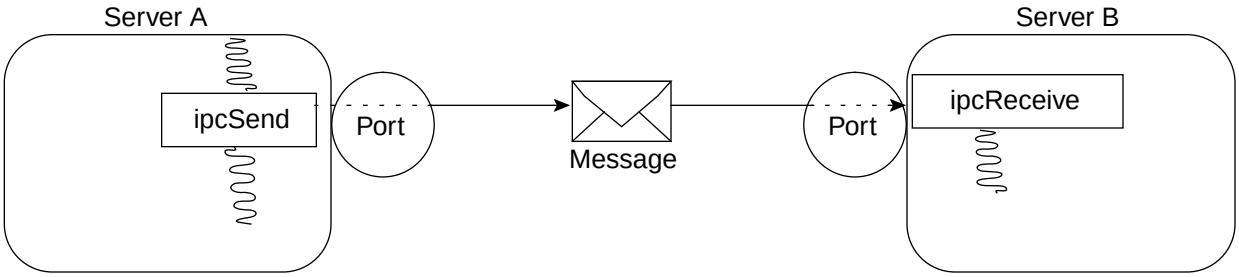
- Asynchronous mode (implemented through ipcSend)

  Asynchronous IPC protocol does not guarantee receipt of the message and does not require a reply (refer again to Figure 12).

Figure 12. Synchronous and Asynchronous IPC Protocol



Synchronous IPC Protocol



Asynchronous IPC Protocol

The microkernel determines whether each IPC is:

- Lightweight IPC (message sent to server at the same site [local memory])

  The microkernel handles lightweight IPC as a procedure call on the same microkernel thread and does not require a context switch. Lightweight IPC is faster and requires fewer system resources.

- Full-weight IPC (message sent to server at a different site)

  The microkernel handles full-weight IPC through messaging. Messaging is supported by E registers in the hardware and requires more system resources than lightweight IPC.

Microkernel Interface Functions

Servers use microkernel interface functions to perform privileged operations. Servers often use microkernel interface functions to process UNICOS system calls. For this reason, microkernel interface functions are sometimes referred to as microkernel system calls.

The following steps demonstrate the relationship between a UNICOS system call and a microkernel interface function under UNICOS/mk. Figure 13 illustrates this example using the same step numbers.

1. An application executes the malloc(3) UNICOS system call to request allocation of additional memory space.

2. This system call causes a context switch as the microkernel connects with the microprocessor. The microkernel recognizes the malloc(3) call as a service that is handled by the PM server and passes the call to an entry point within the PM code.

3. As part of the allocation process, the server makes a rgnAllocate(K) microkernel interface call.

NOTE: For simplicity, many diagrams that illustrate communication between servers intentionally omit context switches to the microkernel.

Figure 13. Example of an malloc(3) System Call under UNICOS/mk

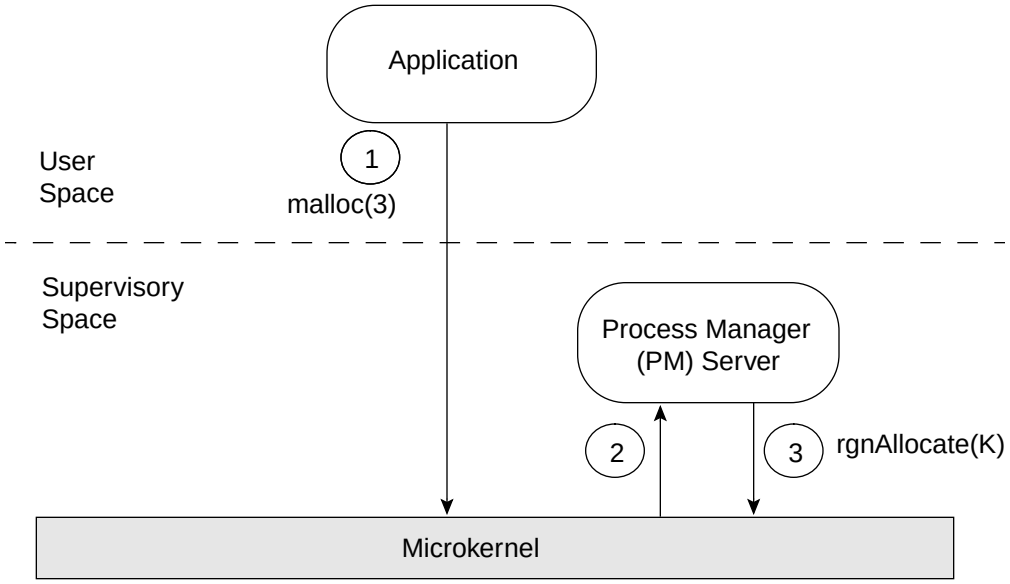


Table 2 lists some of the common microkernel interface functions.

Table 2. Common Microkernel Interface Functions

Function	Definition
intro	Defines terms and error numbers
summary	Summarizes all kernel calls
actorCreate	Creates an actor
actorDelete	Deletes an actor
threadCreate	Creates a thread within an actor
threadDelete	Deletes a thread
threadSuspend	Suspends the execution of a thread
threadResume	Makes a suspended thread eligible for execution
portCreate	Creates a port for an actor
portDelete	Deletes a port
ipcSend	Sends a message to a port without waiting for delivery
ipcReceive	Receives a message on a port
ipcCall	Executes a Remote Procedure Call (RPC) to a port
ipcReply	Replies to an ipcSend(K) or ipcCall(K) message
rgnAllocate	Allocates a new region in an actor's address space
rgnDup	Shares or copies all regions between two actors
rgnFree	Deallocates a region in an actor's address space
vmCopy	Copies memory contents between two actors' virtual address spaces
mutexInit	Initializes a user-defined mutex (semaphore)
mutexGet	Acquires a mutex; sleep if busy
mutexRel	Releases a mutex; wakeup a blocked thread
semInit	Initializes a user-defined counting semaphore
semP	Waits on a semaphore
semV	Signals a semaphore
chorusTod	Gets, sets, or adjusts time-of-day
svMsgHandler	Attaches/detaches a message handler routine to/from a port

File Open Example

Opening a file on a file system requires several servers and at least two PEs. The following example explains how UNICOS/mk may handle an open(2) system call for the CRAY T3E system. Refer to Figure 14 for this example.

1. An application requests a file open by issuing an open(2) system call. The system call causes a context switch between the application and the microkernel.

   NOTE: The microkernel facilitates communication between servers as shown in Figure 13. For simplicity, the microkernel action is omitted from Figure 14.

2. The microkernel recognizes an open(2) system call and points to code in the process manager (PM) server. The PM server checks its open file table to determine whether the file is already open.

3. The PM server issues an IPC to the appropriate file server to request the file open. The file server checks the site's open file table to determine whether it has opened the file for another application.

4. The file server locates the file within the file system and issues an IPC to the disk server with the device address (unit, cylinder, head, sector) of the file information needed for the open.

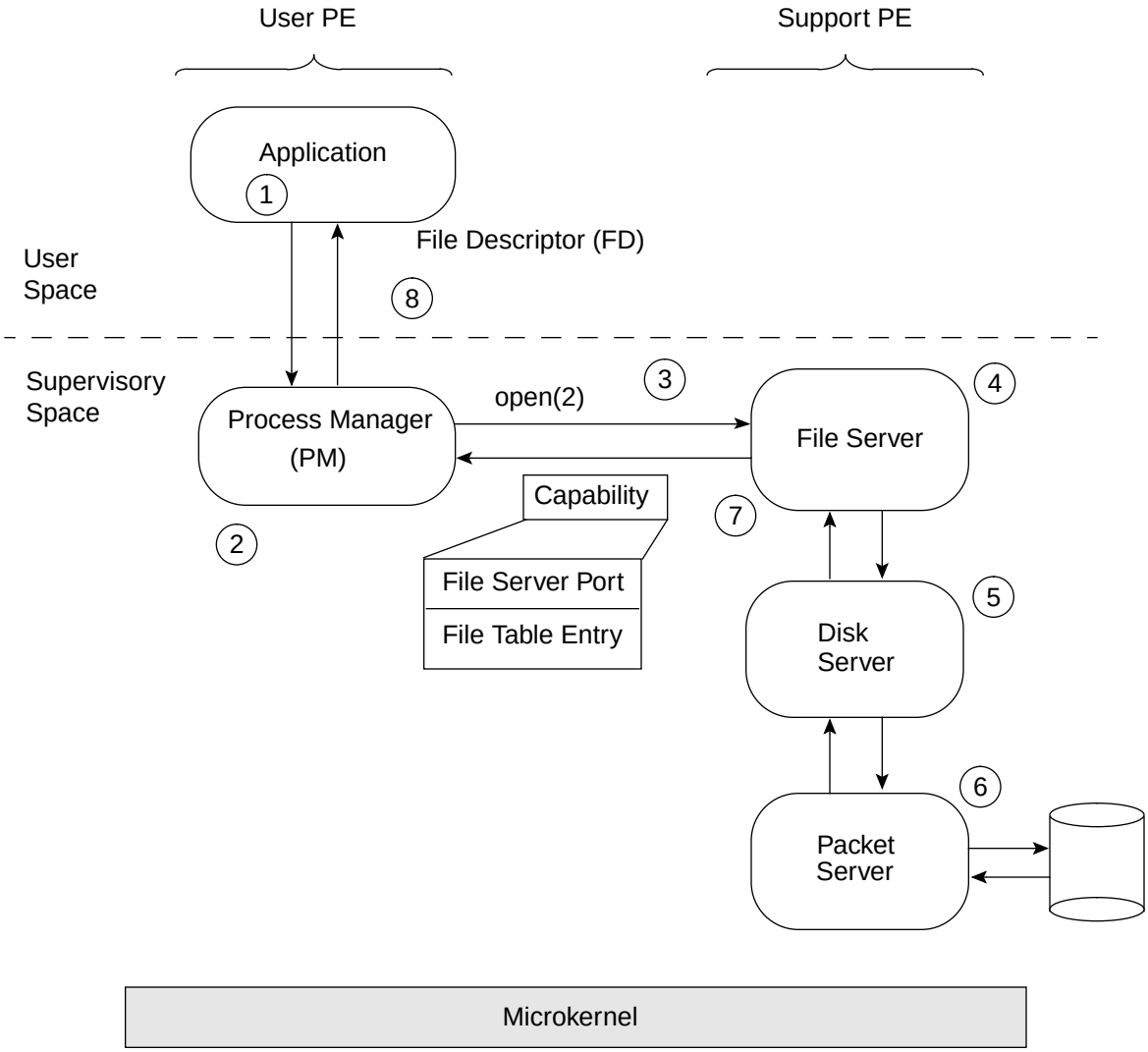
5. The disk server generates the device-dependent information and issues an IPC to the packet server.

6. The packet server forms the packets and sends them to the I/O node of the GigaRing channel.

7. Once the disk responds with the file information that is needed for an open, the packet server uses IPCs to pass the information from the disk to the file server. The file server updates the site's open file table and sends the capability and the file descriptor (FD) to the PM server.

8. The PM server updates the process open file table and passes the FD back to the application. The application can now issue reads and writes to that file by using the FD as the parameter.

Figure 14. File open(2) Example



Additional Information on Server Implementation

The following pages provide additional information on specific servers and their implementation within UNICOS/mk.

Process Manager

The process manager (PM) provides the UNICOS interface to an application. The PM implements a UNICOS interface in addition to the microkernel abstractions. The PM relies on the microkernel to deal with system-specific aspects of the application.

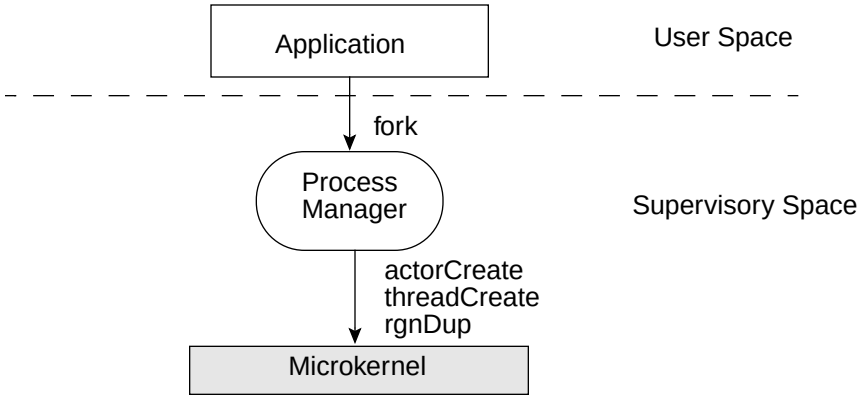
The PM performs the following functions:

- Traps system calls
- Services process-related system calls (for example, fork, exec, wait, exit, sbreak)
- Forwards system calls, including I/O-related system calls (for example, open, read, write, close)
- Implements the functions of a UNICOS process, including process IDs (pids), process groups, signals, etc.

The PM receives and traps all system calls for user applications. When the system call involves a process-related function, the PM services the system call. When the system call is I/O related or socket related, the PM forwards the system call to the appropriate UNICOS/mk server.

Figure 15 shows an example of the PM servicing a fork system call.

Figure 15. PM Servicing a fork System Call

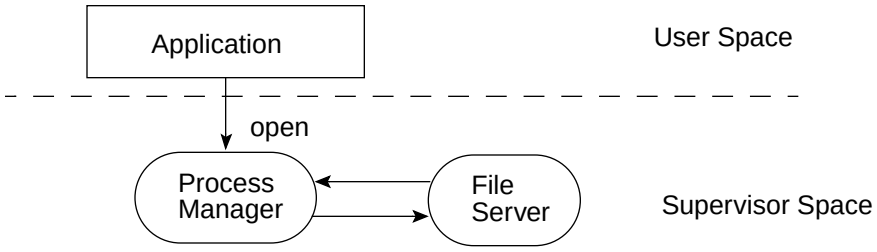


The application issues the traditional UNICOS fork system call (create new process). The process manager creates the necessary CHORUS based abstractions that this system call requires. In this example, these abstractions are an actor (actorCreate), a thread (threadCreate), and memory regions for the second actor (rgnDup).

The fork system call executes completely within the process manager. The microkernel is responsible for the system-specific activity that is required to create a new process (assigning resources to the abstractions).

An open system call is an example of a system call forwarded by the PM (refer to Figure 16). The PM forwards the open system call to the UNICOS file server. The file server executes the request and returns information that the PM uses to find the file on subsequent read requests. The PM stores this information and maps it to a file descriptor.

Figure 16. PM Forwarding an open Request



Local and Global Process Managers

Support of a distributed system requires multiple PMs that form a single system image. This is called *PM distribution* and is implemented by separating the PM into a local process manager (PM) and a global program manager (GPM).

Generally, the PM does all the processing that involves a single process. It also issues requests on functions that involve more than one process to the GPM. The GPM performs operations that consist of multiple processes and maintains information such as the process table, process groups, and sessions.

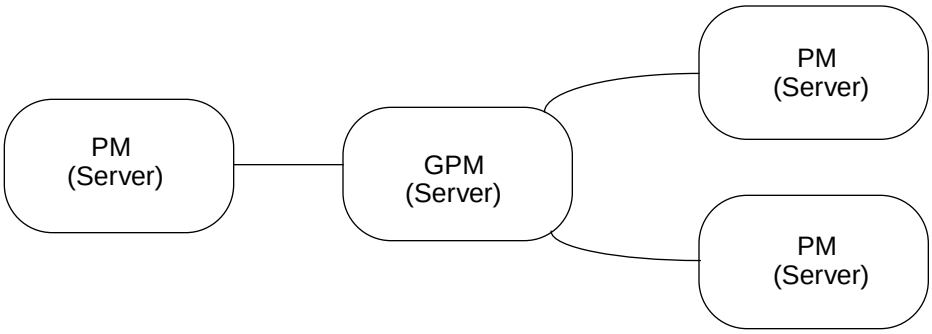
For example, the getpid system call is handled exclusively in the appropriate PM; however, a system call such as kill may require both the PM and the GPM to execute successfully. The PM and GPM can be used in a single-node (PVP) or multiple-node (MPP) environment.

In the single-node (PVP) environment, the PM and GPM reside in one common memory (refer to Figure 17).

Figure 17. Single-node Implementation of PM and GPM

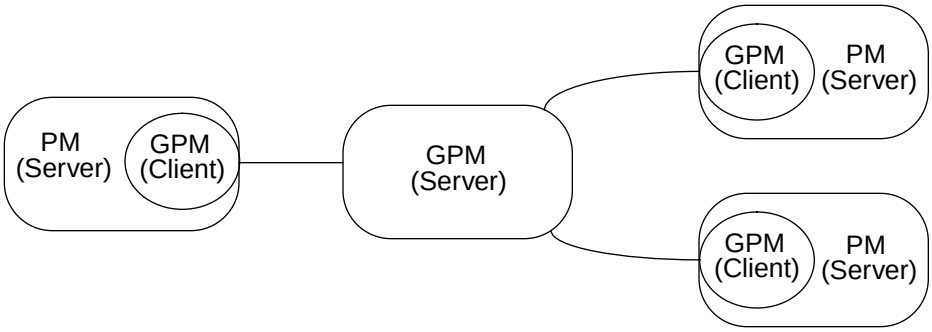
In the multiple-node (MPP) environment, the PM(s) and the GPM are distributed (refer to Figure 18).

Figure 18. Multiple-node Implementation of PMs and GPM



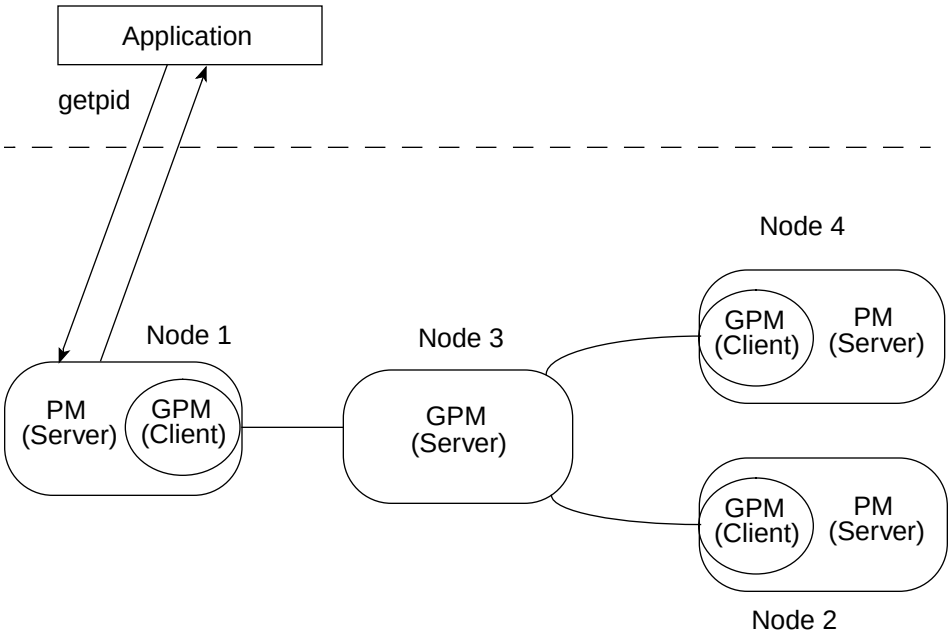
In the multiple-node (MPP) environment, each PM server contains GPM client code that interfaces with the GPM server. (Refer to Figure 19.)

Figure 19. GPM and GPM Client in PM Servers



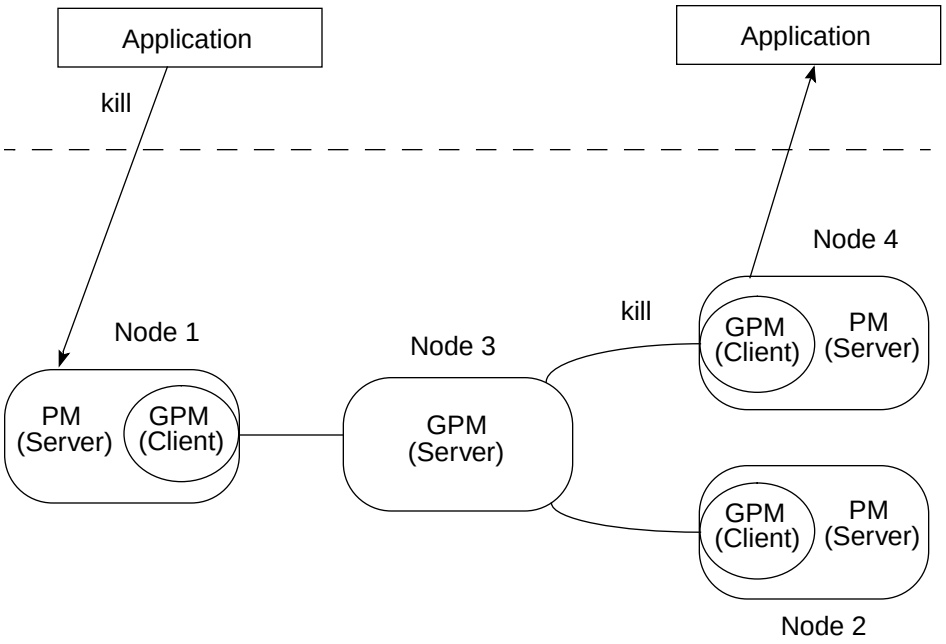
When a getpid system call is issued in a multiple-node environment to the PM on node 1, the PM on node 1 processes the call and returns the process identification (pid) to the caller (refer to Figure 20).

Figure 20. Execution of a getpid Request on a Multiple-node System



When a kill system call is issued to the PM on node 1 in a multiple-node environment, the PM sends the kill system call via the GPM client on node 1 to the GPM server on node 3 (refer to Figure 21). The GPM server has the complete process table. The GPM server uses this table to determine the location of the process that needs to be killed. The GPM server sends the kill system call to the appropriate PM via the GPM client. The PM kills the process.

Figure 21. Execution of a kill Request on a Multiple-node System

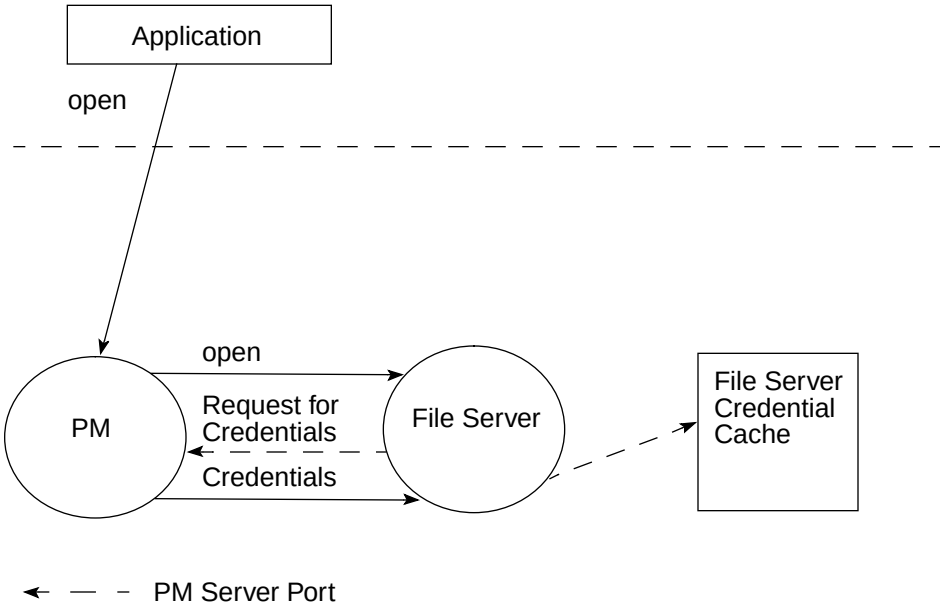


The current implementation of UNICOS/mk uses only one GPM server. This could become a bottleneck for a system with a large number of PMs. To prevent this bottleneck, future implementations of UNICOS/mk will use multiple GPM servers. When the OS uses multiple GPM servers, each GPM server contains a portion of the process table.

Credentials and /proc Server

The PM is also used as a credentials and /proc server. For example, when a file server does not have the credentials for an open system call, the file server signals the PM via the server port (refer to Figure 22). The PM retrieves the credentials from an internal table and sends the credentials to the file server. The file server stores the credentials in its cache and continues processing the open system call.

Figure 22. PM as a Server



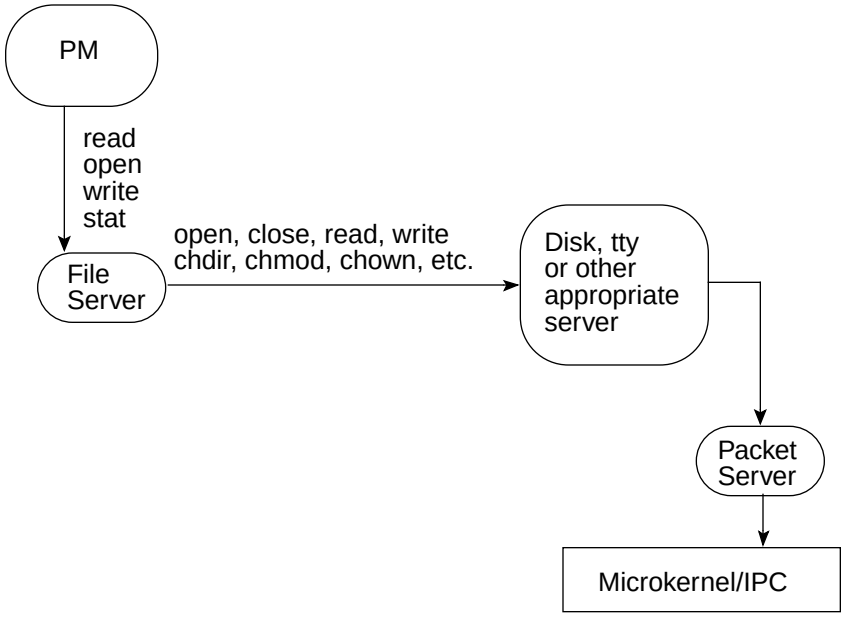
File Server

The file server receives file-related requests from the PM, translates the requests into UNICOS requests, and directs the request to the proper server for execution (refer to Figure 23). Examples of file-related requests are as follows:

- All pathname related requests (open, access, and stat)
- All disk open file requests (read, write, lseek, ioctl, and fcntl)
- Non-disk open file requests (fcntl and fstat)

Most of the code in the UNICOS/mk file server is identical to the file system code used on the current version of the UNICOS system.

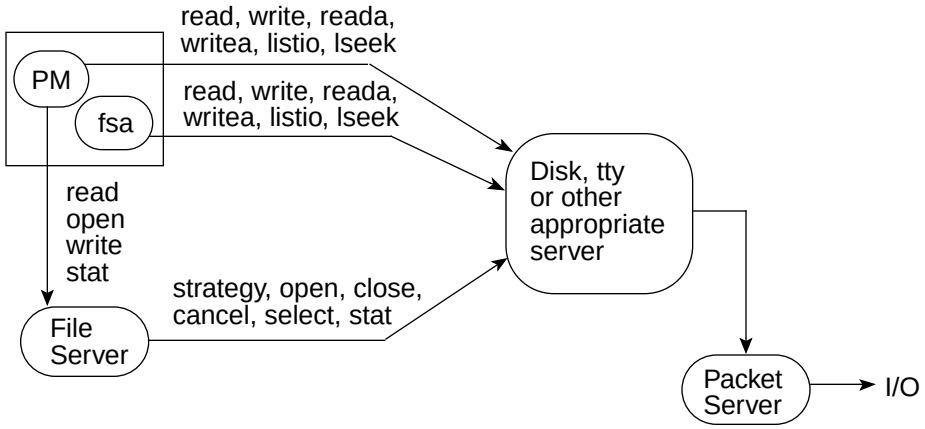
Figure 23. Functions of the File Server



File Server Assistant

The file server assistant (fsa) is a server that implements part of the functionality of the file server to service I/O-intensive applications (refer to Figure 24). The file server assistant allows the user PE to issue read and write disk requests directly to the disk server rather than route them through the file server.

Figure 24. File Server Assistant



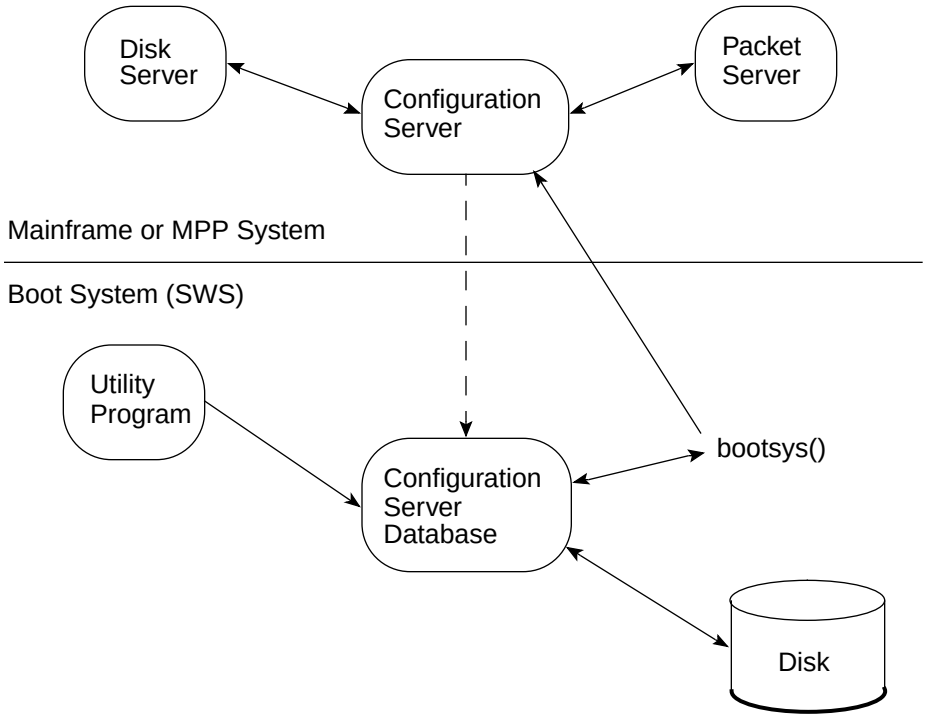
The file server assistant resides on each user PE and distributes basic I/O functions by offloading read, write, and seek requests from remote file servers. The PM forwards all other file-related requests (open, close, etc.) to the file server. This allows the file server assistant to process the read operations and write operations locally, which removes the IPC latency and overhead.

Configuration Server

The configuration server maintains configuration information on the mainframe or MPP system. The configuration server consists of the following components (refer to Figure 25):

- Configuration server
- Configuration server database
- Utility program

Figure 25. Components of the Configuration Server



The configuration server performs the following services:

- Processes requests for information from other servers
- Processes requests to update information from other servers
- Provides notification of changed information to appropriate servers
- Communicates changes to the boot system

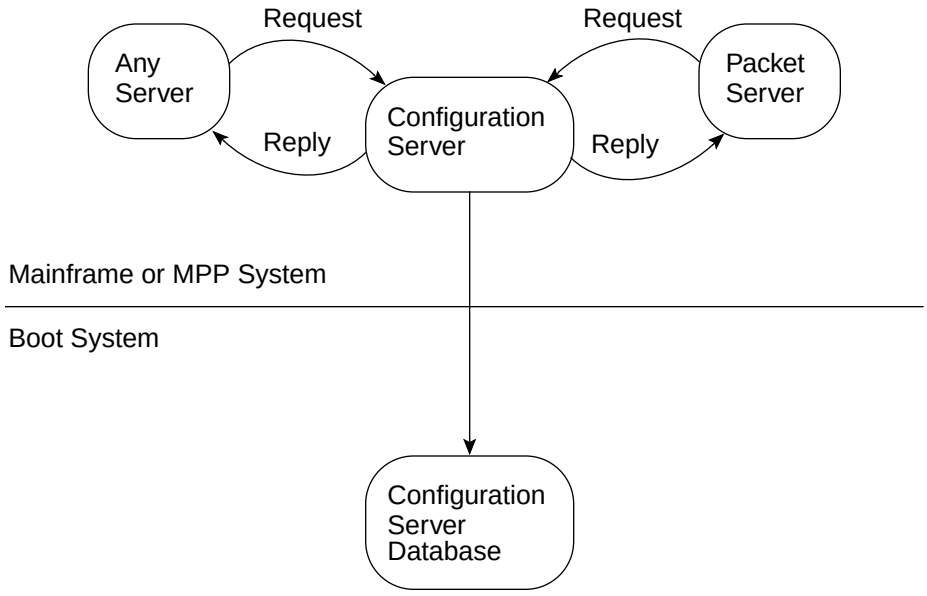
To process requests for information from other servers, the configuration server reads the configuration information from the UNICOS/mk archive and creates tables for this information. The configuration server does not check the information for proper semantics because it assumes the information was checked during the boot process.

Once the configuration tables are created, the configuration server sets up an interface that communicates with other UNICOS/mk supervisor-space servers. All configuration information requests that are made to UNICOS/mk must go through the configuration server.

The configuration server processes requests to update information from other servers by receiving the configuration change from another server and updating the configuration file. The configuration server does not check this new information for errors; it assumes the information is correct. After the configuration server updates the configuration file, the configuration server sends the new configuration information to the configuration server database in the boot system and to other servers that are registered to receive configuration changes. The boot system writes these changes out to the boot-system disk.

The configuration server communicates with other supervisor-space servers through a request-and-reply interface (refer to Figure 26). The request-and-reply interface is synchronous; all information transfers are small messages that contain a single unit of information.

Figure 26. Configuration Server Request-and-reply Interface



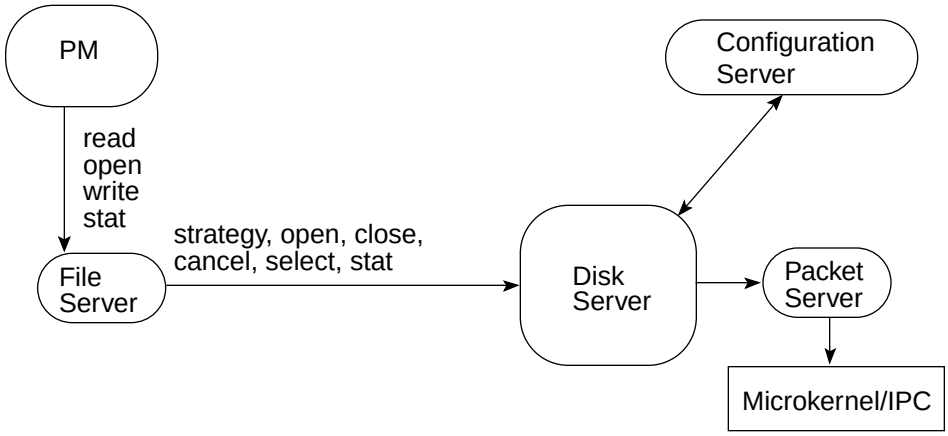
On the boot system, the configuration mechanism contains two parts: the utility program and the configuration server database (refer again to Figure 25). The utility program supplies configuration information from a parameter file and a definition file to the configuration server database. The configuration server database passes this information to the configuration server on the mainframe or the MPP system.

Disk Server

The disk server provides the messaging interface to the file server for block device read/write operations (refer to Figure 27). (The disk server can accept requests from any server; however, the file server makes most of the disk requests.) For full-weight IPC or when the server runs in user mode, the servers make the requests in the form of messages. For lightweight IPC, the servers make requests by calling a function.

NOTE: Raw I/O data flows directly to and from the user process to the I/O device. Buffered I/O data flows directly to and from a buffer cache to the I/O device. The data does not pass through the disk server or the packet server.

Figure 27. Disk Server



The disk server also communicates with the configuration server. During initialization, the disk server sends a request to the configuration server for configuration information. The disk server uses this information to build required tables. The configuration server also passes configuration information to the disk server when a configuration change occurs.

Packet Server

The packet server manages the channels between the Cray Research mainframe and the I/O subsystem. It also provides both a messaging and a direct call interface for other servers that send and receive I/O packets on Cray Research MPP systems.

The packet server has interfaces to the microkernel for interrupt handling and for console I/O. The packet server also has interfaces to other actors. These actors must be registered with the packet server before they can send or receive I/O packets through the packet server.

For lightweight IPC, the source actor and the packet server exchange function addresses during registration. For full-weight IPC, the source actor contains two threads. One thread handles registration and sends packets to the packet server. The other thread issues the ipcCall() function calls to the packet server and waits for a reply.

tty Server

The tty server provides the messaging interface to the process manager and file server for tty character device operations. The file server forwards the following requests to the tty server:

- open
- close
- inactive
- cancel
- select
- inform
- stat

The process manager sends the following requests directly to the tty server:

- read
- write
- reada
- writea
- listio
- lseek
- ioctl
- select

When the tty server receives an open device request, the tty server calls the appropriate device-specific open code. The tty server returns a capability to the file server. The file server stores the capability.

Socket and Network Device Server

The PM forwards any socket operation to the socket server. The socket server forwards the request to the network device server. The network device server then forwards the request through the packet server to the I/O device on the GigaRing channel.

Error Manager

The error manager detects:

- Reportable hardware errors
- Reportable software errors (detected by the hardware)
- Software panics

The error manager attempts to determine which processes, system or user, are affected by an error.

The error manager includes the following components:

- Sanity master
- Error interrupt handler

Sanity Master

The sanity master on the boot PE attempts to detect stalled or hung PEs. To do this, the error manager check-in function of all nonboot PEs reports to the sanity master at a defined interval. When a PE does not check in for two intervals, the sanity master flags the PE as a non-responsive PE and sends a warning message to the system console. If the PE checks in again, the sanity master flags the PE as a responsive PE and sends a message to the system console.

Error Interrupt Handler

The error interrupt handler detects hardware errors that are reported in the following error status registers (detection is subject to mask values supplied by configuration parameters):

- C_ERR[3 : 0]
- M[3 : 0]_ERR[3 : 0]
- R_ERR[4 : 0]
- I_ERR[3 : 0]

NOTE: An error interrupt occurs in the PE that detects the error, which may not be the PE that caused the error.

The error interrupt handler attempts to determine whether the error occurred in a system process or a user process. The error interrupt handler directs fatal error information (most errors are fatal) to the system console and calls the log server to record the errors in /usr/adm/errfile.

In addition, the error interrupt handler calls the log server to record single-bit errors and R-option channel parity errors in /usr/adm/errfile.

NOTE: Use errpt(8) to format and display the data in /usr/adm/errfile. Use crashmk(8) to display the most recent errors that are saved in memory.

Additional Error Handling Information

When an error occurs, UNICOS/mk may perform one or more of the following actions:

- Terminate specific user jobs (performed by process management)
- Display a message on the system console when a PE fails to report (may be hung or stalled)
- Display a message on the system console when a PE that previously failed to report is now reporting
- Perform a hardware prod

Additional Information on the Microkernel

The microkernel provides the following functions:

- Basic system-specific support
- Memory management
- Swap and schedule management

The microkernel also contains a server-to-microkernel interface. This interface consists of a table of valid microkernel calls. The interface references this table whenever a server requests a microkernel service.

System-specific Support

One of the main functions of the microkernel is to provide system-specific support. In other words, the microkernel for the CRAY T3E system is different from the microkernel of the CRAY T90 system. A microkernel hides the specifics of the logic design by providing a virtual system image. To do this, the microkernel exports the actor, thread, memory region, and port abstractions. These abstractions are used by a process manager to implement processes.

Memory Management

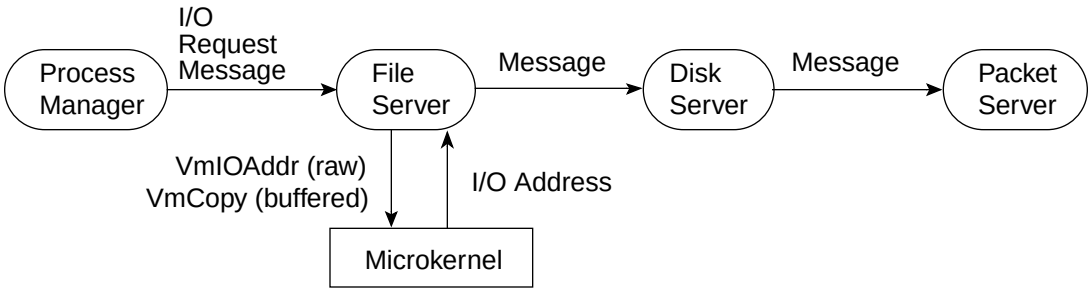
Another function of the microkernel is to manage memory. The memory-management unit of the microkernel consists of tables that indicate how memory is laid out and how it is allocated or swapped.

The memory-management unit performs the following functions:

- Translate the virtual address to a physical address by managing the data translation buffer (DTB) and instruction translation buffer (ITB) within the microprocessor
- Lock and unlock memory to support I/O sequences
- Copy data from user memory to microkernel memory to support a buffered I/O sequence

For example, to retrieve raw data from a disk drive, the file server sends a VmIOAddr function call to the microkernel (refer to Figure 28). This function instructs the memory-management unit to lock memory, retrieve an I/O address from a table, and send this address to the file server. The file server sends the I/O address to the disk server. When the I/O transfer completes, the file server issues a vmUnLock function call to the microkernel. This function call instructs the memory-management unit to unlock memory.

Figure 28. Local Memory I/O Sequence



Another example:

To retrieve buffered data from a disk drive, the file server sends a VmCopy function call to the microkernel. This function instructs the memory-management unit to copy the data from user memory to microkernel memory, retrieve an I/O address from a table, and send this address to the file server. The file server sends the I/O address in a message to the disk server. The disk server sends the message to the packet server. And the packet server sends the message to the I/O controller.

Swap and Schedule Management

The swap operation consists of writing an actor's memory to an external storage device in order to free this memory for use by another actor. UNICOS/mk supports the following swap features:

- The microkernel freezes an actor in preparation to be swapped. To freeze an actor means to halt its threads, wait for any outstanding I/O requests to complete, and block any future references to memory.
- The microkernel transfers memory images to and from secondary storage.
- The microkernel implements the memory scheduling component of the global resource manager.

There are two types of swaps: self swap and forced swap. A self swap occurs when an actor is growing and there is not enough memory to satisfy the actor's request for a new region or the expansion of an existing region of memory. A forced swap occurs when memory is oversubscribed. The global resource manager (grm) determines that an actor needs to be swapped in and chooses which actor must be swapped out.

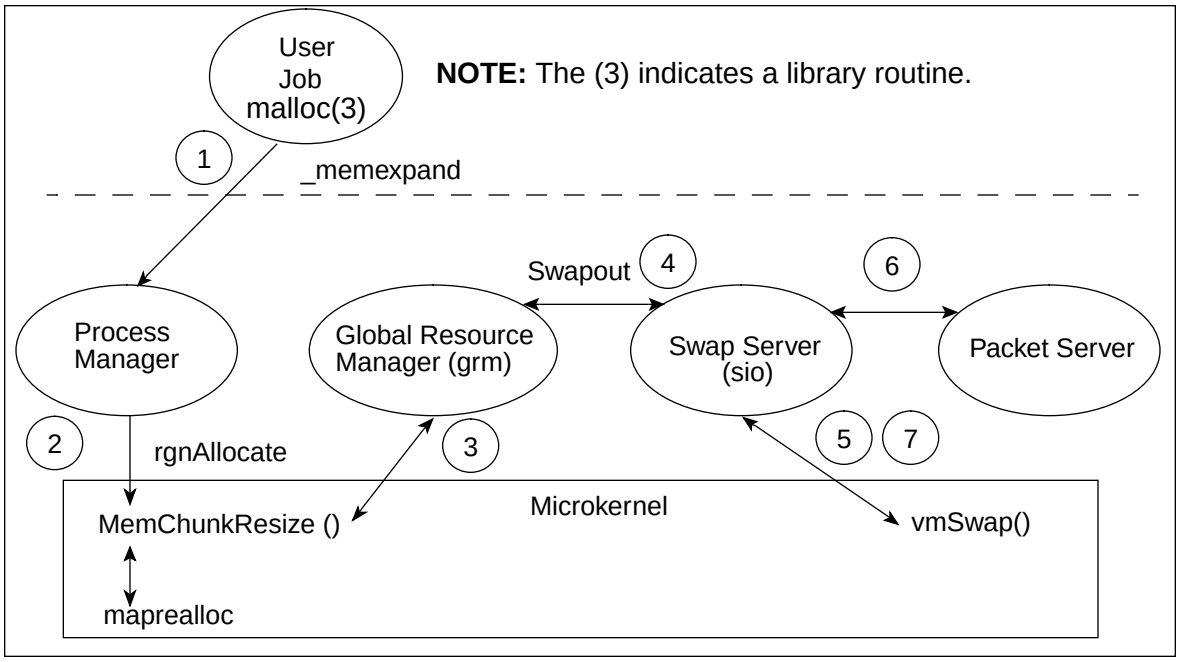
There are two swap procedures: swapout and swapin. The microkernel performs the swapout procedure when an application executes an _memexpand (refer to Figure 29), fork, or exec system call and one of the following conditions exists:

- A process is growing and no expansion is possible.
- A swapped process needs to be swapped in.

grm performs the swapin procedure. It does so when it detects that an actor needs to be swapped in.

NOTE: The malloc library routine makes the system call to change the amount of space that is allocated for the calling process's data segment. The fork system call creates a new process. The exec system call executes a file.

Figure 29. Example of the Swapout Procedure



1. The application executes the malloc library routine that calls the _memexpand system call.

2. The PM sends a rgnAllocate function call to the microkernel. The rgnAllocate function call calls the MemChunkResize function call. And the MemChunkResize function call calls the maprealloc function call. When the maprealloc function call cannot complete because not enough memory is available, the actor cannot grow in place and must be swapped out.

3. The MemChunkResize function call calls grm.

4. grm signals the swap server that a swapout needs to be done.

5. The swap server calls the vmSwap function of the microkernel, which marks the appropriate structures for swapping. The microkernel marks the actors, threads, and local caches that will be swapped. The microkernel fills in the vmLcDesc structure with the physical addresses of the local caches and returns the vmSwap function to the swap server.

6. The swap server uses the information from the vmSwap function to build a message request. The swap server sends the message request to the packet server. The packet server writes the data out to the disk. The packet server notifies the swap server when the entire actor is swapped out.

7. The swap server calls the vmSwap function to notify the microkernel that the actor is swapped out and that the memory reservations can be released.

Index

N

O

P

R

S

T

U

V

W